AIAA-99-4340

USING ABSTRACTION TO CREATE A PORTABLE OBJECT-ORIENTED SIMULATION

P. Sean Kenney and David W. Geyer

Unisys Corporation
NASA Langley Research Center
Mail Stop 169
Hampton, VA 23681

Abstract

A common problem faced in the design of an object-oriented simulation is that there are many different groups of end users of the simulation framework. Each group invariably has a different computer platform on which to run the simulation. In an attempt to satisfy all of these different groups, a simulation framework should be designed to be portable so that it can be operated on as many different platforms as possible. The purpose of this paper is to describe several designs that isolate the framework from the platform dependent services required by a simulation.

Introduction

Like most complex applications, a flight simulation framework requires a large amount of operating system support to perform its tasks. POSIX, the Portable Operating System Interface, is a standard intended to allow an application to move from one operating system to another by simply recompiling it [1]. POSIX is an evolving, growing standard that provides the same interface to system calls on operating systems that support the standard. There are three problems involved with relying on POSIX to provide platform independence:

1) Many platforms do not have POSIX support.
2) Some platforms claim to have POSIX compliance when they only support a minimal POSIX implementation.
3) For a given platform, there may be platform specific services that are not provided in POSIX that could be taken advantage of by a simulation framework.

Until POSIX support is universal and completely supported by most operating systems, another solution to create portable a simulation framework is required.

To provide portability, an object-oriented simulation framework must address three issues. First, the framework must encapsulate the behaviors of operating systems and real-time environments by using abstraction [2] to define interfaces for platform dependent functions. Next, the framework cannot be coupled to any simulation hardware. The framework should isolate complex simulation models from the simulator hardware interfaces. By de-coupling the framework from the hardware, the framework can run on platforms connected to different hardware devices or no hardware devices at all. Finally, if a framework uses a Graphical User Interface (GUI) as a means to control the simulation during execution, the framework cannot be coupled to the GUI. This provides the greatest flexibility for the system as a whole.

The remainder of this paper will discuss each of these issues and how they were dealt with in the Langley

Copyright © 1998 by the authors. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.

Standard Real-Time Simulation in C++ (LaSRS++) Application Framework. LaSRS++ provides a powerful object-oriented framework for dynamic vehicle simulation in real-time [3]. The framework's object-oriented design makes the software extremely flexible, easily maintainable, and provides a high degree of reuse. The LaSRS++ framework currently supports hard real-time simulation on the SGI Onyx and the Convex C3800 platforms. The framework has also been run in a soft real-time mode on SGIs running Irix, Sun workstations running SunOS and Solaris, IBM RS6000s running AIX, and IBM PCs running Linux and Microsoft NT.

The abstractions presented in this paper are the product of an iterative object-oriented design process. The designs provide decoupled, unit-testable, and complete interfaces to platform specific resources. The abstractions are intended to simplify the complexity of porting a simulation framework to new platforms.

Platform Dependent Services

A large number of operating system services are used by a simulation framework to perform real-time simulation. A framework must address scheduling, timing, data sharing, synchronization, I/O, and many other problems. Operating system services provide solutions to these problems but directly tie the framework to a platform. A framework must use a design that isolates operating system implementation details from the framework. Such a design allows the framework to use timers, schedulers, shared memory, semaphores, and other operating system resources in a portable fashion. LaSRS++ uses an elegant design employing the Abstract Factory, Bridge, and Singleton design patterns to completely isolate the framework from the implementation details required to use operating system services [4].

The design will be described using two common operating system services, shared memory and semaphores. Both of these services are commonly used in simulation frameworks, and the required system calls differ greatly from platform to platform.

Shared Memory

Most modern operating systems provide a means to map memory into the memory spaces of two or more processes at once thereby allowing the processes to share data. This mapping is known as shared memory. The system calls used to create and access a shared memory segment vary from platform to platform. Using object-oriented techniques, it is possible to present users on all platforms with a common interface to shared memory but allow the actual low-level implementation to vary according to the platform being used. This is accomplished by using several different object-oriented design patterns. Design patterns describe simple and elegant solutions to specific problems in object-oriented software design.

Figure 1 uses the Unified Modeling Language (UML) to show the class diagram for the shared memory interface software.

Bridge Pattern

The Bridge pattern decouples an abstraction from its implementation.[2] This design uses the Bridge pattern to isolate client code from the platform specific details of a specific implementation. The approach is to have clients use an abstraction object that forwards its public member function class to a hidden platform specific implementation object. The abstraction object uses the implementation object through the pure polymorphic interface defined by the abstract implementation base class. In this case, a shared memory interface object (SharedMemory) interacts with a platform specific shared memory implementation object (SharedMemoryImpl) through a polymorphic interface. An appropriate concrete implementation class is defined for each platform on which the simulation framework is executed upon. The appropriate concrete implementation object for a given platform is selected at run-time.

Abstract Factory Pattern

The Abstract Factory pattern is used to create the correct instance of the platform specific shared memory implementation object at run time. This creational pattern provides an interface for creating families of related objects without specifying their concrete classes. In this case, the family of related objects are the plat-

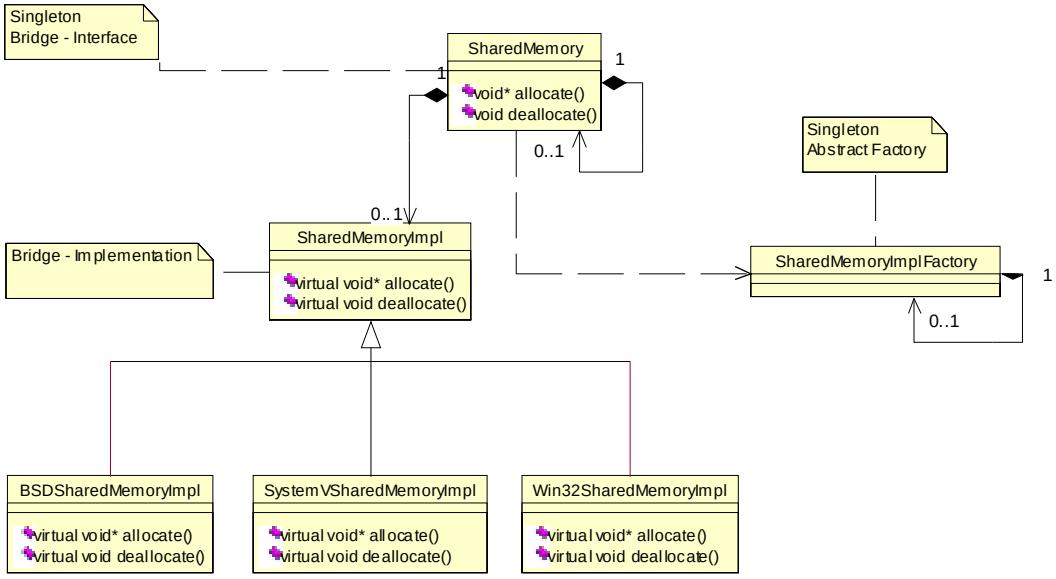

Figure 1 – Shared Memory Interface

form specific implementation objects. The abstract factory object (SharedMemoryImplFactory) contains knowledge of the specific platform that is being used. The constructor for the shared memory interface object invokes the makeSharedMemoryImpl member function of the abstract factory object. This member function uses knowledge of the specific platform to return the appropriate implementation object for the given platform. This specific implementation object is stored as a hidden attribute of the shared memory interface object.

Singleton Pattern

The Singleton creational pattern is used whenever it is necessary to ensure a class only has one instance, and provide a global point of access to the single instance. The shared memory interface class and the abstract factory class are implemented as singletons to provide a global point of access for creating and accessing a shared memory space.

Design Advantages

This design has several advantages that address the issues of portability and maintainability:

1. *Client code is de-coupled from the platform specific shared memory implementation details.* This de-coupling allows changes in the shared memory implementation classes to have no impact on the client code; i.e., the client code does not need to be recompiled if the implementation changes. By coupling client code only to a common interface, the client code becomes platform independent. This allows client code to be portable to any platform supported by the shared memory interface class.
2. *Only a single class needs to be written to support a new platform.* Adding support for a new platform involves two steps:
   (a) Write a new shared memory implementation class that interfaces with the platform specific shared memory routines
   (b) Add to the Shared memory implementation abstract factory the capability to create the new shared memory implementation object

3. *Creation of platform specific objects can be isolated to a single abstract factory object.* The shared memory interface class only references the abstract shared memory base class. All concrete implementation details are confined to the implementation abstract factory.

4. *The design is generic and can be applied to other operating system services.* The following section describes how the design was applied to another important operating system service - semaphores.

It is important to note that the concrete implementation classes cannot usually be compiled on any platform other than the one the class is intended. Makefile directives are the most common method of dealing with this issue.

Semaphores

A semaphore is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented every time a thread or process "acquires" the semaphore object and incremented every time a thread or process "releases" the semaphore. When the count reaches zero, no more threads or processes can successfully acquire the semaphore and will not proceed until it can. Semaphores are useful in controlling shared resources and synchronizing multiple threads or processes.

Figure 2 shows the class diagram for the semaphore interface software as implemented in LaSRS++. The semaphore interface software design uses the same patterns as found in the shared memory interface software. The only difference is that the System V implementation class is an abstract class rather than a concrete class. Two new concrete classes derive from the SystemVSemaphoreImpl class. These concrete classes provide the implementation details for these two platforms that are different among the two System V Unix platforms.

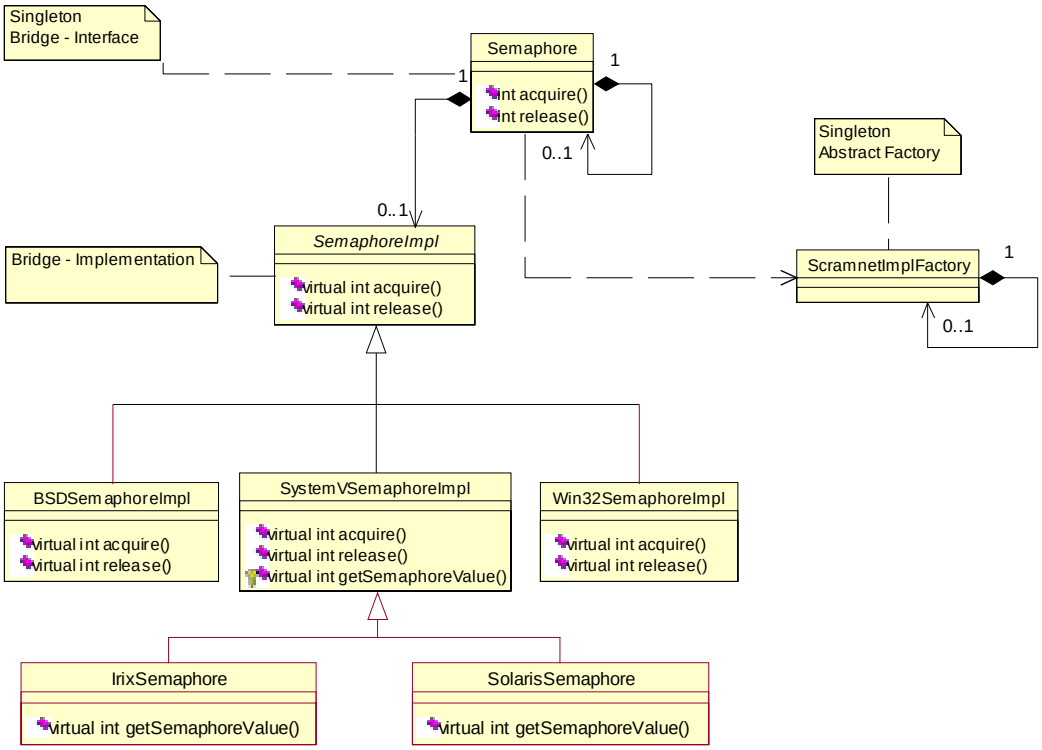

Figure 2 – The Semaphore Interface

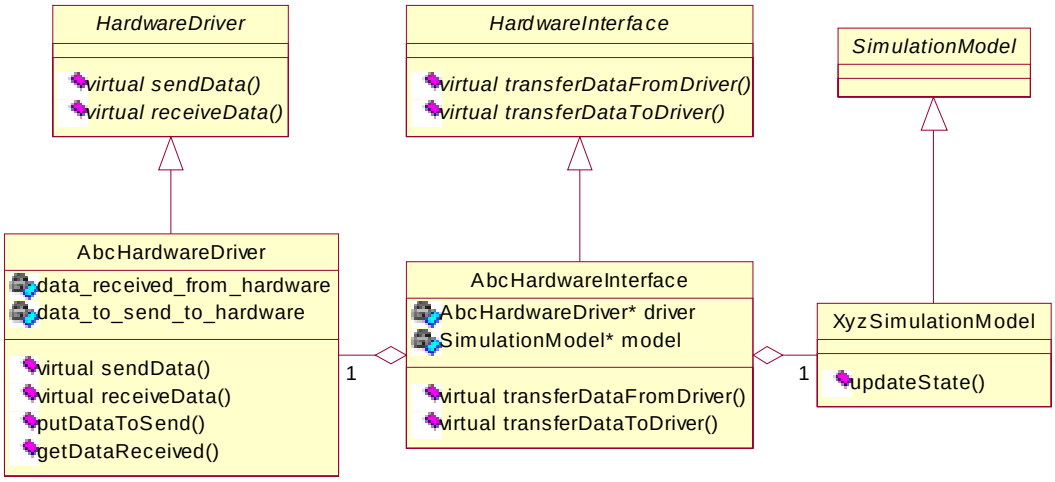

Figure 3 - Drivers, Interfaces, and Simulation Models

As described above, the design is very generic. LaSRS++ employs the design to handle several other operating system services.

The Hardware Abstraction

The hardware abstraction is composed of three main components: drivers, interfaces, and builders [5]. The drivers are the classes that actually transmit and receive data with the simulator hardware devices, the interfaces are communication classes that pass data between a simulation model and a driver, and the builders construct all of the appropriate drivers and interfaces as needed. Figure 3 demonstrates the relationships between the drivers, the interfaces, and the simulation models.

Mediator Pattern

The Mediator design pattern keeps classes from referring to each other explicitly and encapsulates how the set of classes interact.[2] The strongest asset of the Mediator design pattern is that it completely decouples the two classes from each other. It should be used whenever two classes are unrelated but need to communicate with each other.

Drivers

The driver classes are the classes that actually transmit and receive data with the simulator hardware devices. They typically contain buffers to hold the data that is transferred with the hardware and member functions to access or modify the data buffers. In the above diagram, the class AbcHardwareDriver defines the two virtual functions declared in the abstract class HardwareDriver. These virtual methods send and receive data. The class also defines methods that access and modify data transferred to and from the hardware. The driver class therefore provides the abstract interface of HardwareDriver and an interface specific to the "Abc" hardware. The driver classes are an implementation of the Bridge design pattern.

Interfaces

Interfaces are communication classes that pass data between the driver and the simulation models. The interface class also performs any manipulation of the data before transferring the data to its destination. The interface class is essentially a one way or two way data pump between the driver and the simulation model. In the illustration above, the AbcHardwareInterface is given a reference[†] to the AbcHardwareDriver and the XyzSimulationModel when it is instantiated. The class would then use the two references to transfer data between the two classes when appropriate. The interfaces are a variation of the Mediator design pattern.

[†] Unless stated otherwise, reference refers to both the reference and pointer types in C++.

Builders

Builder classes construct all of the appropriate drivers as requested by the user and construct all of the corresponding interfaces required by the simulation models. The builder classes provide an interface for creating the driver and interface classes without other simulation classes having knowledge of the particular concrete classes. The builder classes are implemented using the Abstract Factory design pattern.

Design Advantages

The advantages of this design are:

1. *The simulation models are completely decoupled from the hardware driver classes.* This allows the models to be tested with or without simulator hardware. The behavior of the simulation models with different inputs can be fully tested offline allowing comprehensive analysis of performance without using valuable simulator hardware resources. Once the performance of a model has been validated, the hardware inputs/outputs can be used to validate the model's performance in the simulation. This minimizes the validation required of new models. The decoupled simulation models also remain portable. A polymorphic class hierarchy allows different computation models to be incorporated into the simulation and use the existing hardware interface class without modification. The model classes may be exported to other sites without requiring any modifications for use. A model imported from another site can be "wrapped" in a class that has the interface required by the existing hardware interface, thereby quickly assimilating the new computational model into the simulation.

2. *Modifications to simulator hardware only require a change to the driver class.* Many hardware devices receive major and minor modifications over their lifetimes. Minor modifications are often changes to software, buffer sizes, etc. and require little or no change to the software used to communicate with the device. Major modifications may require a significant change to the software used to communicate with the device however. Because the driver encapsulates all of the code involved with direct hardware communication, the simulation is completely isolated from the modifications.

3. *The hardware driver classes can be unit tested*. Any modifications to a driver class can be tested without the simulation model. A diagnostic program can be written that uses the driver to communicate with the hardware. The diagnostic program serves two functions. It can be used to test any changes made to the driver program and it can be used to verify the operation of the hardware prior to use by the simulation. Software configuration management eases the burden of testing the new driver class by allowing the user to verify that the hardware is operating correctly with a previous version of the driver before testing the new version. (This is usually not possible when the hardware has been modified).

4. *The driver and/or interface class may be used to emulate the hardware*. Often a simulation uses real-world hardware like a flight management computer to assist in research or testing. A software emulation of a hardware device can be placed in either the driver or interface class to allow the simulation to perform necessary communications with the emulated hardware when the real hardware is unavailable. The hardware emulation may also be modified to conduct research experiments.

5. *Changes to the models can not affect the communication between a driver and it's respective hardware*. A modification to a simulation model may result in bad data being transmitted to a hardware device, but it can not cause a connection loss or crash if the hardware interface class properly limits the data being sent to the hardware device.

6. *The hardware interface classes are generally very trivial*. The classes simply use the accessor and modifier methods of the driver class and the simulation model to transmit data. Any calculations required when manipulating the data can easily be verified through testing.

7. *The hardware interface classes can often be reused by different simulation models without modification*. If the models share a common base class and the hardware interface only uses a reference to the base simulation model then no modification is required for use with different simulation models.

8. *Hardware driver and interface classes are only used on platforms that support their use*. If the framework is run on a platform that has no support for a particular hardware device, then the abstract factory builder class will not attempt to create these classes. The builder classes are the only class dependent upon hardware driver and interface classes. Because only the builder classes are dependent on the hardware classes, the framework is both maintainable and portable.

9. *Only two classes need to be written to support a new hardware device*. Adding support for a new hardware device involves three steps:

   (a) Write a new hardware driver class that communicates with the new hardware device

   (b) Write a new hardware interface class that transfers data between the driver and the simulation model as needed

   (c) Add the new classes to the abstract factory builder classes to create the new driver and interface as needed.

GUI Isolation

Simulation frameworks often use GUIs to control and/or monitor simulation states during execution. To ensure portability and maintainability, a framework cannot be coupled to the GUI. This provides the greatest flexibility for the system as a whole.

Two designs are available that allow a GUI to control a framework with minimal coupling. The first design involves using a shared memory segment as a means to allow the GUI access to simulation states [6]. This method requires the framework to instantiate all objects that are manipulated by the GUI into shared memory. The GUI can then attach to the shared memory segment and manipulate objects using their public methods. The design has the drawback that it requires framework classes to be instantiated into shared memory.

Another design involves creating a second thread to create and manage a GUI [7]. In this design a framework class must create and start the new thread. The GUI is then able to monitor and/or control simulation states because it shares the same memory as the simulation framework. Due to the fact that many GUI toolkits are not thread-safe, the GUI must be designed to only allow a single thread to execute function calls to the GUI toolkit.

It should be noted that in both of these designs, the framework is de-coupled from the GUI while the GUI is directly coupled to the framework. All communication between the framework is unidirectional. The GUI monitors and controls simulation states. The framework is completely unaware of the presence of the GUI.

Determining which GUI library or toolkit to use is another difficult issue. Three options are available to a framework designer:

1) Use the libraries that came with a platform/operating system/compiler.

2) Use commercial or free libraries from other sources.

3) Develop a library suite for each of the platforms needed in house.

Each option has its advantages and disadvantages. Using the libraries commonly found on a platform is an inexpensive solution but can limit portability. For example, most UNIX platforms have native support for X windows and ship with other graphical libraries like Xt and Motif. If Motif is selected as the GUI library for a framework, then the framework is limited only to UNIX platforms that also support Motif.

Using commercial libraries or free libraries is an excellent alternative to using the libraries shipped with a platform. Problems may arise however in finding a product that supports all off the target platforms. Support problems may also impact the development of framework features. If errors are found in these products, development of new framework features or

framework corrections may not be possible until after the vendor addresses the problem. Another problem arises when the platform requires on operating system upgrade and an operating system compatible library is not available.

Developing a library in house provides a framework staff with the greatest flexibility but has tremendous resource requirements until the libraries are full implemented.

The GUI toolkit used by LaSRS++ has undergone several transitions since the framework was first created. Originally the framework was only run only on UNIX machines. The Motif library was used to develop a GUI to control the simulation while it was executing. The shared memory GUI design was used to allow the GUI to monitor and control the framework. Because Motif is not object-oriented, the GUI code base quickly became extremely large and awkward to maintain. The Motif library was abandoned in favor of an object-oriented toolkit called *Amulet*. The Amulet base GUI also used the shared memory design. Amulet is a GUI toolkit developed at Carnegie-Mellon University and supports almost every flavor of UNIX, Microsoft Windows, and even the Macintosh. The library was also freely downloadable over the Internet. Unfortunately DARPA funding of the Amulet project ended, and the product is no longer being developed or supported by its creators. The loss of Amulet support was a mixed blessing – a new toolkit was selected that required the GUI to be rewritten yet again. The new toolkit provides a better object-oriented foundation than Amulet and has better performance. Currently, LaSRS++ uses the free GUI toolkit called *gtk--*, a GNU software package. The library is also object-oriented and supports most UNIX platforms, Microsoft Windows, and the Macintosh. The LaSRS++ framework was also modified to use the thread based GUI design. Because of the framework is de-coupled from the GUI, users have the option to run with either the Amulet based GUI or the gtk-- based GUI. This demonstrates the importance of framework/GUI isolation. While the gtk-- based GUI was being developed, other framework users where continued to use the older GUI. This affords users the convenience of transitioning to the new GUI according to their timetable.

Conclusions

Design patterns were used to maximize code reuse while encapsulating implementation details within platform specific classes. The Bridge, Factory, and Singleton patterns were used extensively throughout the design to achieve a portable framework. The Factory pattern allows the framework to know about abstract classes and nothing about the subclasses that define the platform specific operations. The pattern establishes an interface for creating subclasses without requiring any knowledge of the subclasses. The Bridge pattern decouples an abstraction from its implementation. This allows the abstraction and implementation to vary independently. The pattern provides flexibility in that the abstraction and implementation are in separate class hierarchies. The Singleton pattern ensures only one instance of a class exists and provides a global point of access to it. Essentially, the above patterns allow the entire body of code that composes a framework to be unaware of which platform the framework is operating on.

Because the hardware interface is abstracted away in a manner that keeps the actual simulator hardware communications hidden from the rest of the framework, the framework is a robust, portable, and easy to maintain simulation system. Continual reuse of the hardware abstraction ensures that new hardware interfaces can be easily added into the framework and that these interfaces can be easily tested and debugged with or without the hardware.

Isolation of the framework from the GUI provides the greatest flexibility when moving the framework to different platforms. In selecting a GUI library for use with a simulation framework, a designer must weigh the pros and cons of the aforementioned options and select the most appropriate solution for that particular environment.

Although the designs presented in this paper were originally designed to support flight simulation at NASA Langley Research Center, the designs could be used in any object-oriented framework to heighten reuse, portability, and maintainability.

Bibliography

[1] Bill O. Gallmeister. *POSIX.4 Programming For The Real World*. O'Reilly & Associates, Sebastpol, California, 1995.

[2] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, California,1994.

[3] Richard A. Leslie, et al. *LaSRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft*. Paper Number AIAA-98-4529, August, 1998.

[4] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[5] P. Sean Kenney, et al. *Using Abstraction To Isolate Hardware In An Object-Oriented Real-Time Simulation*, Paper Number AIAA-98-4533, August, 1998.

[6] Michael Madden, et al. *Constructing a Multiple-Vehicle, Multiple-CPU Using Object-Oriented C++*. Paper Number AIAA-98-4530, August, 1998.

[7] David Geyer, et al. *Managing Memory Spaces In An Object-Oriented Real-Time Simulation*, Paper Number AIAA-98-4532, August, 1998.

[8] Bruce Eckel. *Thinking in C++*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[10] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, Reading, Massachusetts, 1996.

[11] Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice-Hall, Englewood Cliffs, 1995.

[12] Scott Meyers. *Effective C++*. Addison-Wesley, Reading, Massachusetts, 1992.

[13] Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, Massachusetts, 1996.

[14] David R. Musser, Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996.

[15] Terry Quatrani, *Visual Modeling With Rational Rose and UML*, Addison Wesley, Reading, Massachusetts, 1998.

[16] Pierre-Alain Muller. *Instant UML*. Wrox Press Ltd., 1997. ISBN 1-861000-87-1.